PERI Autotuning of PFLOTRAN

Jacqueline Chame[1], Chun Chen[2], Mary Hall[2], Jeffrey K. Hollingsworth[3], Kumar Mahinthakumar[4], Gabriel Marin[5], Shreyas Ramalingam[2], Sarat Sreepathi[4], Vamsi Sripathi[4], Ananta Tiwari[6]

[1]USC/ISI, Marina del Rey, CA 90292
[2]University of Utah, Salt Lake City, UT 84112
[3]University of Maryland, College Park, MD 20742
[4]North Carolina State University, Raleigh, NC 27695
[5]Oak Ridge National Laboratory, Oak Ridge, TN 37831
[6]San Diego Supercomputing Center, San Diego, CA 92093

E-mail: mhall@cs.utah.edu

Abstract. In response to the enormous and growing complexity of today's high-end systems, the Performance Engineering Research Institute (PERI) is working toward automating portions of the performance tuning process by developing an *autotuning framework*. Our framework employs empirical techniques to identify the best-performing version of a computation among a search space of possible implementations. This paper describes application of PERI performance tools to demonstrate performance gains on production scientific codes on DOE's leadership-class systems. This paper examines the use of PERI tools in automatic and semi-automatic tuning of PFLOTRAN, which models subsurface flow in groundwater.

1. Introduction

As we prepare for near-term tens of petaflops systems and future exascale systems, the increasing architectural complexity will correspondingly increase the difficulty of realizing the performance potential of these architectures. Historically, the burden of achieving high performance on new platforms has largely fallen on application scientists. To improve their productivity, we would like to provide performance tools that are (largely) automatic, a long-term goal commonly called *autotuning*. This goal encompasses tools that analyze a scientific application, both as source code and during execution, generate a space of implementation options, and search for a near-optimal performance solution. The Performance Engineering Research Institute (PERI) is formalizing a performance tuning methodology used by application developers and automating portions of this process. Autotuning is one of the three aspects of performance tuning that are the focus of PERI, in addition to performance modeling and application engagement. In the context of these three aspects, the goal of PERI is to migrate automatic and semi-automatic prototypes into practice for a set of important applications. Although a relatively new technology, this paper examines PERI efforts in applying autotuning to PFLOTRAN, a case study DOE application that models subsurface flows in groundwater.

This document focuses on the PERI autotuning strategy (Section 2) and tuning of PFLOTRAN (Section 3), followed by a discussion of future work (Section 3).

2. Tuning PFLOTRAN

The U.S. Department of Energy (DOE) is interested in studying the effects of geologic sequestration of CO_2 in deep reservoirs and migration of radionuclides and other environmental contaminants in groundwater. Modeling of subsurface flow and reactive transport is necessary to understand these problems [4]. PFLOTRAN [3] [6] is a highly scalable subsurface simulation code that solves multiphase groundwater flow and multicomponent reactive transport in three-dimensional porous media. It employs domain-decomposition parallelism and makes extensive use of PETSc (Portable, Extensible Toolkit for Scientific Computation) [5] numerical solvers and distributed data structures. It uses MPI (Message Passing Interface) through PETSc for interprocess communication and the parallel HDF5 (Hierarchical Data Format 5) [1] library to perform simulation I/O. Initial performance analyses of PFLOTRAN with respect to scalability, single-node performance, and I/O performance revealed that PFLOTRAN exhibited markedly different behavior with respect to these three performance metrics on the Cray XT5 and IBM BG/P architectures. Thus, PFLOTRAN was chosen as the architecture tiger team code in 2009 for PERI's targeted modeling and autotuning activities.

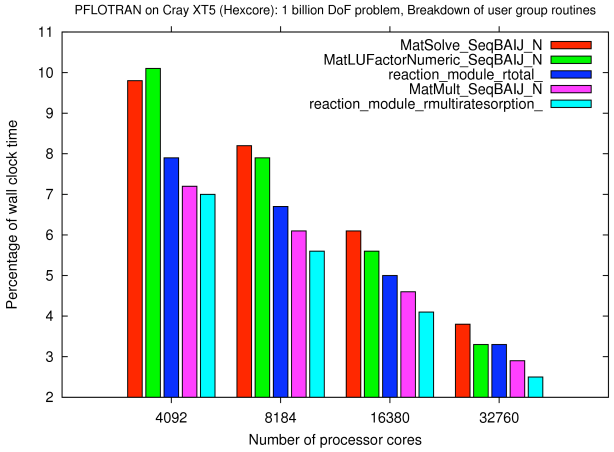

Figure 1. Dominant user routines on Cray XT5.

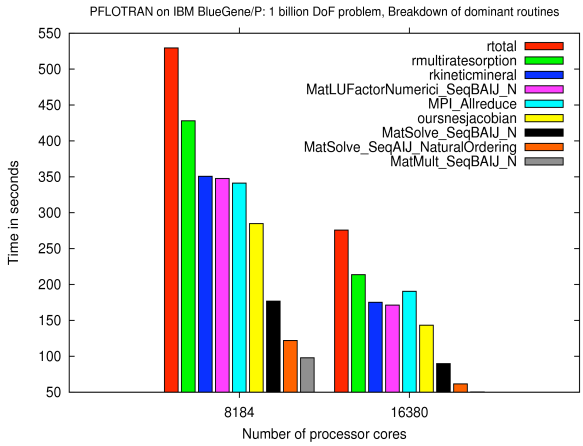

Figure 2. Dominant user routines on BG/P.

To further understand the performance issues facing PFLOTRAN, we gathered execution profile data using both Cray XPAT (a vendor tool) and HPC Toolkit (a PERI tool) on the Jaguar Cray XT5 at Oak Ridge National Laboratory. The timing breakdown of predominant user routines in PFLOTRAN for a 1 B DOF run on the Cray XT5 and BG/P are shown in Figures 1 and 2, respectively. We discovered that PFLOTRAN spends more than 25% of its sequential execution time in three library routines from the PETSc library. Two of these routines that constitute 17% of the execution time are achieving less than 5% of theoretical peak performance on a node on Jaguar. Additionally, the application spends more than 6% in a single-loop nest computation within the PFLOTRAN procedure *reaction_module_rtotal* that also achieves less than 5% theoretical peak performance. We chose not to optimize the PETSc LU factorization, *MatLUFactorNumeric*, as it performs at nearly 20% peak on both architectures. Therefore, we focused our autotuning efforts on the following three functions:

- *MatMul_SeqBAIJ_N*, a PETSc routine that computes a block sparse matrix-vector multiplication;
- *MatSolve_SeqBAIJ_N*, a PETSc triangular solve function that solves the system $\boldsymbol{A\,x = b}$, given an $\boldsymbol{LU}$ factored matrix $\boldsymbol{A}$, also in a block sparse representation; and
- *Reaction_module_rtotal (RTOTAL)*, a PFLOTRAN routine that calculates the contribution of aqueous equilibrium complexity to residual and Jacobian functions for Newton-Raphson iterations.

Autotuning PETSc functions. To optimize the two PETSc routines, we start with a clean implementation of the key loop nests, with no pointer arithmetic and eliminating the copies at the beginning of the functions. In this way, we focus on just the computation in the functions. Both routines are using a block sparse matrix representation, where the input matrix is a collection of dense blocks of a fixed size. The *N* on the end of the function names denotes that the fixed block size is a parameter to the function. The value of *N* is equal to 15 for many of the production problems solved in PFLOTRAN, representing the degrees of freedom in PETSc and the number of chemical species in PFLOTRAN. Figure 3 summarizes the autotuning process for triangular solve using two integrated PERI tools: the CHiLL transformation and code generation system and the Active Harmony framework for navigating the optimization search space [9].

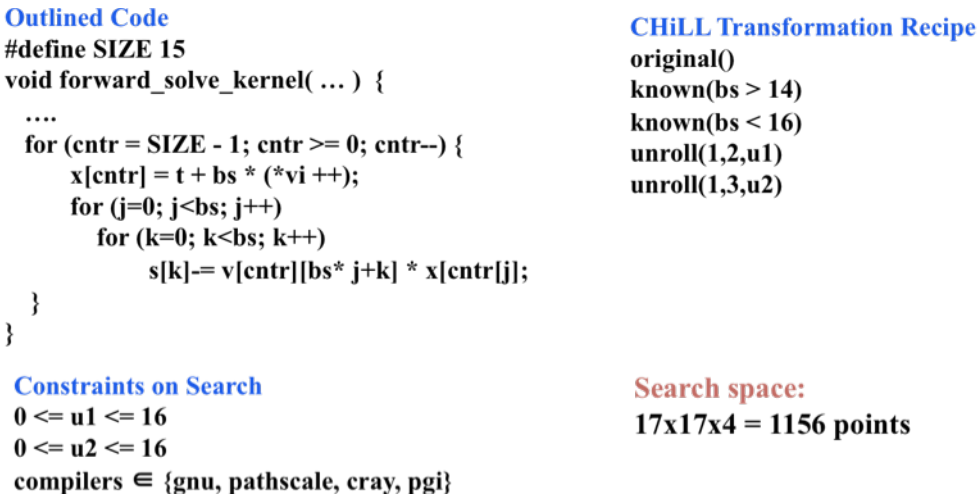

Figure 3. Triangular solve autotuning process.

In Figure 3 we show the forward solve portion of the triangular solve code. On the top right, we show a transformation recipe provided to the CHiLL system to specialize the function for a block size of 15 and unroll the loop nest computation to expose instruction-level parallelism and improve register allocation. The unroll factors are constrained so that they are less than the block size. Then we employ the Active Harmony system to search the space of possible unroll factors for the loop nest to identify the one that leads to the best performance. The search space consists of over 1,000 points, but Active Harmony employs a parallel rank order search to limit the number of points that must be considered. These autotuning experiments were performed on Jaguar as well, and a summary of performance results is shown in Table 1.

Table 1. Autotuning performance summary for PETSc triangular solve.

Compiler	Original	Active Harmony			Exhaustive		
	Time	Time	(u1,u2)	Speedup	Time	(u1,u2)	Speedup
pathscale	0.58	0.32	(3,11)	1.81	0.30	(3,15)	1.93
gnu	0.71	0.47	(5,13)	1.51	0.46	(5,7)	1.54
pgi	0.90	0.53	(5,3)	1.70	0.53	(5,3)	1.70
cray	1.13	0.70	(15,5)	1.61	0.69	(15,15)	1.63

The experiment involved trying all combinations of unroll factors for inner and outer loops, and the four compilers available on Jaguar (the four rows) to achieve speedup over the original execution time (the second column). The search space was small enough that it was possible to exhaustively search these options. The exhaustive search results are shown in the rightmost three columns: the execution time, two unroll factors, and speedup over the original execution time. By comparison, columns three through five show the comparable results with Active Harmony, showing slightly different unroll factors and performance. Active Harmony results are obtained by evaluating, on average, less than 50 points, which amounts to less than 5% of the total points in the search space. Another perhaps surprising result is how such a simple piece of code achieves very different performance on the same architecture using different compilers. The optimization strategy and results are similar for the matrix vector multiply. We specialized for different column sizes (105, 90, 75, 60) and row size 15. Since the search space is fairly small, we did an exhaustive parameter sweep (two unroll factors) for the most dominant column sizes 105 and 90 using the pgi compiler. On average, the best CHiLL-generated code-variant performs 1.5X faster than the original implementation.

Autotuning RTOTAL. The PFLOTRAN routine we optimized has a single loop nest with several loops nested inside, each with a very small number of iterations (typically 2, 3, or 4) corresponding to the number of aqueous complexes associated with each primary species. Again the optimization strategy is to specialize the loop nest for the small number of iterations and then unroll loops to expose instruction-level parallelism, register reuse, and reduce branching. Using this strategy, we achieve an overall performance speedup of 27% for just this function.

Other PFLOTRAN enhancements. Initial profiling studies on Cray XT5 indicated that I/O is a major bottleneck for scaling PFLOTRAN beyond 32K cores. This is primarily because parallel I/O libraries such as HDF5 that rely on MPI-IO do not scale well beyond 10K processor cores, especially on parallel file systems (like Lustre) with single point of resource contention. PERI's I/O optimization efforts led to a two-phase I/O approach at the application level where a set of designated processes participate in the I/O process by splitting the I/O operation into a communication phase and a disk I/O phase. The designated I/O processes are created by splitting the MPI global communicator into multiple sub-communicators. The root process in each subcommunicator is responsible for performing the I/O operations for the entire group and then distributing the data to rest of the group. This approach resulted in over 25X speedup in HDF I/O read performance and 3X speedup in write performance for PFLOTRAN at over 100K processor cores on the ORNL's Cray XT5 systems [7][8].

Final results. After autotuning, each of these routines was sped up by nearly a factor of two. Additional I/O tuning described above resulted in a 40-fold speedup of the initialization phase, 4-fold improvement in the write stage, and a 5-fold improvement of total simulation time on 90,000 cores.

Limits on further improvement. In our investigation, we also used modeling [2] to understand the principal data reuse patterns in PFLOTRAN and to identify opportunities for enhancing reuse in cache. Table 2 presents the top eight program scopes where data is long-lived (measured in L3 cache misses carried by each scope) and therefore may not have locality in cache. A program scope *C* is said to be carrying a reuse to datum *D* when two consecutive accesses to *D* are both taking place after entering *C* and before exiting *C*, and no other program scope is entered more recently than *C* and still active at the time the two data accesses execute. In some cases, the loop carrying the reuse can be found locally where the data is accessed, for example, when a loop iterates over the inner dimension of an array. Other times, some outer loop, which might be in a different routine, causes the application to access the same data repeatedly, for example, in consecutive time steps. In general, the farther removed the carrying scope from the place where the data is accessed, the harder it is to shorten the reuse. The results show that 58% of all L3 cache misses are the product of long data reuse across iterations of various PETSc linear solvers or across iterations of the program's main time-step loop and are inherent due to dependences between iterations inside the iterative solvers. Another 21% of all L3 cache misses are caused by data reuse inside the solver routines between the initialization loops

and the main solver loops and are very difficult to improve without rewriting the solvers. Thus, at least 80% of all L3 cache misses are extremely difficult to optimize away, while another 10% not shown in the table are distributed very thinly over many loops, which means each code transformation would cause marginal improvement. By using data reordering we were able to eliminate almost all L3 cache misses and TLB2 misses carried by the initialization loop of the Jacobian evaluation routine, namely, 5% of L3 cache misses and 12% of TLB2 misses, but this optimization is not applicable in all cases.

Table 2. Top program scopes responsible for long data reuse distances.

Program Scope	Carried L3 Misses (%)	Carried TLB2 Misses (%)	Comments
Loop at ls.c [181-249]	33.4%	21.6%	Main loop of nonlinear system solver using line search
Loop at bcgs.c [70-121]	19.8%	12.3%	Main loop of biconjugate gradient squared solver
rtjacobian_	9.3%	9.1%	Jacobian main routine
KSPSolve_BCGS	7.5%	4.6%	Biconjugate gradient squared solver main routine
Loop at baijfact2.c [5134-5199]	6.0%	3.7%	Loop in LU factorization routine
Loop at reactive_transport.F90 [2428-2478]	5.0%	13.5%	Initialization loop for Jacobian evaluation
Loop at timestepper.F90 [385-492]	4.6%	3.1%	PFLOTRAN time-step loop
KSPSolve	4.3%	2.6%	Linear system solver routine

3. Summary and Future Work

We reported on a PERI activity to tune the performance of PFLOTRAN through a combination of manual and automated tuning. While we made significant improvements in PFLOTRAN performance, it was through a combination of tuning the computation and optimizing I/O and communication. We also identified limits on performance gains that will require significant modifications to the PFLOTRAN implementation. The experience with tuning PETSc inspired a project to develop a generalized framework that permits specialization of PETSc for a specific *execution context*, the combination of an application, its dataset, the target architecture and the node-level compiler. In further experiments with the PFLOTRAN-invoked PETSc kernels, we have shown even higher performance gains than reported above, and we will integrate these into the application.

References

[1] HDF5. http://www.hdfgroup.org/HDF5/, date accessed: June 2011.

[2] Marin, G., and J. Mellor-Crummey (2008). "Pinpointing and exploiting opportunities for enhancing data reuse," in *Proceedings of the 2008 IEEE Intl. Symposium on Performance Analysis of Systems and Software*, April.

[3] Mills, R. T., C. Lu, P. C. Lichtner, and G. E. Hammond (2007) "Simulating subsurface flow and transport on ultrascale computers using PFLOTRAN," *J. Phys.: Conf. Ser.*, 78, doi:10.1088/1742-6596/78/1/012051.

[4] Mills, R. T., G. E. Hammond, P. C Lichtner, V. Sripathi, G. Mahinthakumar, and B. F Smith (2009). "Modeling subsurface reactive flows using leadership class computing," *J. Phys: Conf. Ser.*, 180(1):012062, 2009.

[5] PETSc Home page. http://www.mcs.anl.gov/petsc, date accessed: June 2011.

[6] PFLOTRAN Home page. http://ees.lanl.gov/pflotran/, date accessed: June 2011.

[7] Sripathi, V., G. E. Hammond, G. Mahinthakumar, R. T. Mills, P. H. Worley, and P. C. Lichtner (2009). "Performance analysis and optimization of parallel I/O in a large scale groundwater application on the Cray XT5," Poster presentation, Supercomputing 2009, Portland, Oregon, Nov. 12-16.

[8] Sripathi, V. (2010). "Performance analysis and optimization of parallel I/O in a large scale groundwater application

on petascale architectures," M.S. thesis, North Carolina State University, June. http://www.lib.ncsu.edu/resolver/1840.16/6098.

[9] Tiwari, A., C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth (2009). "A scalable autotuning framework for compiler optimization," in *Proceedings of the International Parallel and Distributed Processing Symposium*, May.